Seuss: Decoupling responsibilities from static methods for fine-grained configurability

Schwarz, Niko; Lungu, Mircea; Nierstrasz, Oscar

Published in:
Journal of Object Technology

DOI:
10.5381/jot.2012.11.1.a3

Publication date:
2012

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Seuss: Decoupling responsibilities from static methods for fine-grained configurability

Niko Schwarz[a] Mircea Lungu[a] Oscar Nierstrasz[a]

a. University of Bern

Abstract Unit testing is often made more difficult by the heavy use of classes as namespaces and the proliferation of static methods to encapsulate configuration code. We have analyzed the use of 120 static methods from 96 projects by categorizing them according to their responsibilities. We find that most static methods support a hodgepodge of mixed responsibilities, held together only by their common need to be globally visible. Tight coupling between instances and their classes breaks encapsulation, and, together with the global visibility of static methods, complicates testing. By making dependency injection a feature of the programming language, we can get rid of static methods altogether. We employ the following semantic changes: (1) Replace every occurrence of a global with an access to an instance variable; (2) Let that instance variable be automatically injected into the object when it is instantiated. We present Seuss, a prototype that implements this change of semantics in Smalltalk. We show how Seuss eliminates the need to use class methods for non-reflective purposes, reduces the need for creational design patterns such as Abstract Factory and simplifies configuration code, particularly for unit tests. We present benchmarks showing that Seuss introduces a 34 % additional memory cost, and runs at 53 % speed, without any optimizations.

Keywords Dependency Injection, Empirical Studies, Object-oriented Design

1 Introduction

Class methods, which are statically associated to classes rather than instances, are a popular mechanism in object-oriented design. Java and C#, for example, provide static methods, and Smalltalk provides "class-side" methods, methods understood by classes, rather than their instances. 9 of the 10 most popular programming languages

Niko Schwarz, Mircea Lungu, Oscar Nierstrasz. *Seuss: Decoupling responsibilities from static methods for fine-grained configurability.* In Journal of Object Technology, vol. 11, no. 1, 2012, pages 3:1–23.
doi:10.5381/jot.2012.11.1.a3

listed by TIOBE provide some form of static methods.[1] In most of these languages, classes are the key mechanism for defining namespaces. For this reason, static methods offer a convenient mechanism for defining globally visible services, such as instance creation methods. As a consequence, static methods end up being used in practice wherever globally visible services are needed.

Unfortunately this common practice leads callers of static methods to implicitly depend on the classes that provide these static methods. The implicit dependency on static methods complicates testing. That is because unit tests should test only the unit under test, and mock [MFC01] the rest of the application. However, mocks can hardly be plugged in from the outside when static methods are called from within the unit under test, and thus hard-wired into the code. Clearly, a better alternative to static method calls is needed. In order to be able to argue for a credible alternative to static method calls, we need to better understand the need for static methods in the first place.

Classes are known to have both meta-level and base-level responsibilities [BU04]. To see what those are, we examined 120 static methods, chosen at random from SqueakSource, a public repository of open source Smalltalk projects. We found that while nearly all static methods inherited from the system are reflective in nature, only few of the user-supplied methods are. Users never use static methods to define reflective functionality.

Dependency injection is a design pattern that shifts the responsibility of resolving dependencies to a dedicated dependency *injector* that knows which dependent objects to inject into application code [Fow02, Pra09]. Dependency injection offers a partial solution to our problem, by offering an elegant way to plug in either the new objects taking over the responsibilities of static methods, or others required for testing purposes. Dependency injection however introduces syntactic clutter that can make code harder to understand and maintain.

We propose to regain program modularity while maintaining code readability by introducing dependency injection as a language feature. *Seuss* is a prototype of our approach, implemented by adapting the semantics of the host language. Seuss eliminates the need to abuse static methods by offering dependency injection as an alternative to using classes as namespaces for static services. Seuss integrates dependency injection into an object-oriented language by introducing the following two semantic changes:

1. Replace every occurrence of a global (such as a class literal) with an access to an instance variable;

2. Let that instance variable be automatically injected into the object at instantiation time.

Seuss cleans up class responsibilities by reserving the use of static methods for reflective purposes. Furthermore, Seuss simplifies code responsible for configuration tasks. In particular, code that is hard to test (due to implicit dependencies) becomes testable. Design patterns related to configuration, such as the Abstract Factory pattern, which has been demonstrated to be detrimental to API usability [ESM07], become unnecessary.

[1] TIOBE Programming Community Index for January 2011, `http://www.tiobe.com`. Those 10 languages are Java, C, C++, PHP, Python, C#, (Visual) Basic, Objective-C, Perl, Ruby. The outlier is C, which does not have a class system.

This paper is an extension of previous work presented at the TOOLS conference 2011 [SLN11]. It adds the following contributions:

1. The description of the implementation in Section §5 is further detailed.
2. The performance and memory consumption of Seuss are evaluated in Section §6
3. The discussion Section §8 is expanded. Among other things, we outline possible optimizations in Section §8.2.
4. The development using Seuss is detailed in Section §7

Structure of the article. In Section §2 we analyze the responsibilities of static methods and establish the challenges for reassigning them to suitable objects. In Section §3 we demonstrate how Seuss leads to cleaner allocation of responsibilities of static methods, while better supporting the development of tests. In Section §4 we show how some creational design patterns in general and the Abstract Factory design in particular are better implemented using Seuss. In Section §5 we go into more details regarding the implementation of Seuss. In Section §6 we evaluate the memory consumption and time performance of Seuss. In Section §7 we discuss development using Seuss in more detail. In Section §8 we discuss the challenges for statically-typed languages, and we summarize issues of performance, human factors and security. In Section §9 we summarize the related work and we conclude in Section §10.

2 Understanding class responsibilities

Static methods, by being associated to globally visible class names, hard-wire services to application code in ways that interfere with the ability to write tests. To determine whether these responsibilities can be shifted to objects, thus enabling their substitution at run-time, in subsection 2.1 we first analyze the responsibilities static methods bear in practice. Then in subsection 2.2 we pose the challenges facing us for a better approach.

2.1 Identifying responsibilities

We follow Wirfs-Brock and Wilkerson's [BW89] suggestion and ask what the current responsibilities of static methods are, for that will tell us what the new classes should be.

We determine the responsibilities following a study design by Ko *et al.* [KMA04]. Their study identifies six learning impediments by categorizing insurmountable barriers encountered by test subjects. The authors of the paper independently categorize the impediments and attain 94 % agreement.

We examined 120 static methods and classified their responsibilities from a user's point of view. For example, a static method that provides access to a tool bar icon would be categorized as providing access to a resource, regardless of how it produced or obtained that image. We chose 95 projects uniformly at random from SqueakSource,[2] the largest open source repository for Smalltalk projects. We then selected uniformly at random one static method from the latest version of each of these projects. To avoid biasing our analysis against framework code, we then added 25 static methods selected

[2] `http://www.squeaksource.com/`

uniformly at random from the standard library of Pharo Smalltalk,[3] as shipped in the development environment for developers.

Of the 120 methods selected, two were empty. We randomly chose another two methods from SqueakSource to replace them. Two subjects then categorized the 120 methods independently into the categories, achieving 83 % agreement. We then reviewed the methods that were not agreed upon. Most disagreements were due to lack of knowledge of the exact inner workings of the API they were taken from. After further review, we placed them into the most appropriate subcategory. Figure 1 presents an overview of the distribution of methods in categories.

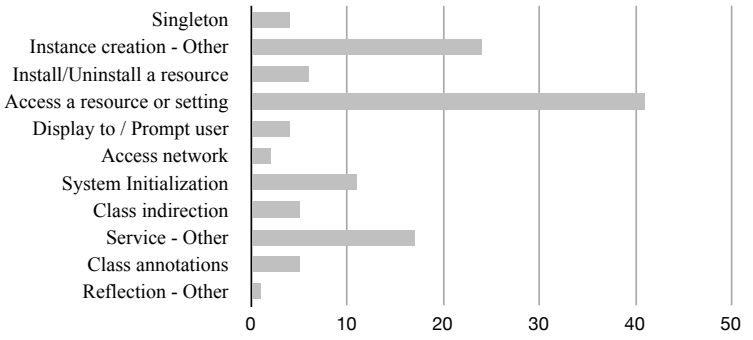


Figure 1 – The classification of responsibilities of the static methods from a user's point of view in our study

We identified the following three umbrella categories: *Instance creation*, *Service* and *Reflection*, each further subdivided into subcategories. Whenever a method did not fit into any of the subcategories, we marked it as "other".

Instance creation

(28 of 120) Instance creation methods create new instances of their own class. They are subdivided as follows.

Singleton. (4 of 28) These methods implement the singleton pattern [GHJV94] to ensure that the instance is created only once.

Other. (24 of 28) Some methods provided default parameters, some simply relayed the method parameters into setters of the newly created instance. Only 3 methods did anything more than setting a default value or relaying parameters. These three methods each performed simple computations on the input parameters, such as converting from minutes to seconds, each no longer than a single line of code.

Services

(86 of 120) Service methods provide globally available functionality. They often serve as entry points to an API. We have identified the following sub-categories.

Install/uninstall a resource. (6 of 86) By resource, we mean a widely used object that other parts of the system need to function. Examples of installable resources

[3] `http://pharo-project.org/`

that we encountered are: packages of code; fonts; entries to menus in the user interface.

Access a resource or setting (41 of 86) These methods grant access to a resource or a specific setting in a configuration. Complex settings resemble resources, hence one cannot easily distinguish between the two. Examples include: a status object for an application; the packet size of headers in network traffic; default CSS classes for widgets; a sample XML file needed to test a parser; the default lifetime of a connection; the color of a GUI widget.

Display to/prompt user (4 of 86) Examples: showing the recent changes in a versioning system; opening a graphical editor.

Access network (2 of 86) These methods grant access to the network. Examples: sending an HTTP put request; sending a DAV delete request.

System initialization (11 of 86) These methods set the system status to be ready for future interactions. Examples: setting operation codes; setting the positions for figures; asking other system parts to commence initialization.

Class indirection (5 of 86) These return a class, or a group of classes, to provide some indirection for which class or classes to use. For example, method compilerClass in Object returns Compiler, in order to allow subclasses to choose another compiler for their methods.

Other (17 of 86) Other responsibilities included: converting objects from one class to another; taking a screenshot; sorting an array; granting access to files; starting a process; mapping roles to privileges; signaling failure and mailing all packages in a database.

Reflection

(6 of 120) Unlike methods that offer services, reflective methods on a class are by their nature tightly coupled to instances of the class. We have found the following sub-categories.

Class Annotations. (5 of 6) Class annotations specify the semantics of fields of their class. All the examples we examined were annotations interpreted by Magritte [RDK07], a framework for adapting an applications model and metamodel at run-time.

Other. (1 of 6) One method provided an example on how to use the API.

2.2 Challenges

Out of the 120 static methods we have analyzed, only 6 belonged naturally and directly to the instances of that class, namely the reflective ones. All other responsibilities can be implemented in instance methods of objects tailored to these responsibilities.

We conclude that static methods are defined in application code purely as a matter of convenience to exploit the fact that class names are globally known. Nothing prevents us from shifting the responsibilities of non-reflective static methods to regular application objects, aside from the loss of this syntactic convenience. In summary the challenges facing us are:

- to shift static methods to be instance responsibilities,
- while avoiding additional syntactic clutter, and
- enabling easy substitution of these new instances to support testing.

In the following we show how Seuss, our dependency injection framework, allows us to address these challenges.

3 Seuss: moving services to the instance side

We would like to turn misplaced static methods into regular instance methods, while avoiding the syntactic clutter of creating, initializing and passing around these instances. Dependency injection turns out to be a useful design pattern to solve this problem, but introduces some syntactic clutter of its own. We therefore propose to support dependency injection *as a language feature*, thus maintaining the superficial simplicity of global variables but without the disadvantages. Dependency injection furthermore shifts the responsibility of injecting dependent variables to a dedicated *injector*, thus enabling the injection of objects needed for testing purposes. Let us illustrate dependency injection in an example.

In the active record design pattern [Fow02, p. 160 ff], objects know how to store themselves into the database. In the SandstoneDB implementation of active record for Smalltalk [Leo08] a Person object can save itself into the database as in Figure 2.

```
user := Person firstName: 'Ramon' lastName: 'Leon'.
user save.
```

Figure 2 – Using the active record pattern in SandstoneDB

The code of the save method is illustrated in Figure 3. (The actual method is slightly more complicated due to the need to handle further special cases.)

```
save
    ↑ self critical: [
        self onBeforeSave.
        isFirstSave
            ifTrue: [Store storeObject: self]
            ifFalse: [Store updateObject: self].
        self onAfterSave.
    ]
```

Figure 3 – The save method in SandstoneDB, without dependency injection.

The save method returns the result of evaluating a block of code in a critical section (self critical: [...]). It first evaluates some "before" code, then either stores or updates the state of the object in the database, depending on whether it has previously been saved or not. Finally it evaluates the "after" code.

In the save method, the database must somehow be referenced. If the database were an ordinary instance variable that has to be passed during instance creation, the code for creating Person objects would become cluttered. The conventional workaround is to introduce static methods storeObject: and updateObject: to encapsulate the responsibility of connecting to the database, thus exploiting the global nature of the

Store class name, while abusing the mechanism of static methods for non-reflective purposes.

Unfortunately, testing the save method now becomes problematic because the database to be used is hard-wired in static methods of the Store class. There is no easy way to plug in a mock object [MFC01] that simulates the behavior of the database for testing purposes.

The dependency injection design pattern offers a way out by turning globals into instance variables that are automatically assigned at the point of instantiation. We add a method to Person that declares that Person is interested to receive a Store as an instance variable during instance creation by the runtime environment, rather than by the caller, as seen in Figure 4. Afterwards, instead of accessing the global Store (in upper case), save is re-written to access instance variable store (in lower case; see Figure 5).

```
store: anObject
    <inject: #Store>
    store := anObject
```

Figure 4 – Person declares that a Store should be injected upon creation.

```
save
    ↑ self critical: [
        self onBeforeSave.
        isFirstSave
            ifTrue: [store storeObject: self]
            ifFalse: [store updateObject: self].
        self onAfterSave.
    ]
```

Figure 5 – The save method from SandstoneDB rewritten to use dependency injection does not access the globally visible class name Store.

In the example in Figure 5, we also see that Person does not ask specifically for an instance of a class Store. It only declares that it wants something injected that is labeled #Store. This indirection is beneficial for testing. Method storeObject: may pollute the database if called on a real database object. Provided that there is a mock class TestStore, we can now inject instances of that class rather than real database objects in the context of unit tests.

Avoiding cluttered code by language alteration. The dependency injection pattern introduces a certain amount of clutter itself, since it requires classes to be written in an idiomatic way to support injection. This clutter manifests itself in terms of special constructors to accept injected objects, and factories responsible for creating the injected objects. Seuss avoids this clutter by incorporating dependency injection as a language feature. As a consequence, the application developer may actually write the code as it is shown in Figure 3. The semantics of the host language are altered so that the code is interpreted as shown in Figure 5.

In Seuss, what is injected is defined in configuration objects, which are created in code, rather than in external configuration files. Therefore, we can cheaply provide configurations tailored for specific unit tests. Figure 6 illustrates how a unit test can now test the save method without causing side effects. The code implies that the storeObject: and updateObject: methods are defined on the instance side of the

TestStore class.

```
testing := Configuration bind: [ :conf | conf bind: #Store to: TestStore new].
user :=~(Injector forConfiguration: testing get: #User).

user firstName: 'Ramon' lastName: 'Leon'.
user save.
```

Figure 6 – Unit test using dependency injection. The injector interprets the configuration, and fills all dependencies into user, including the TestStore.

Typically, a developer using dependency injection has to explicitly call only one injector per unit test, and only one for the rest of the application, even though the injector is active during every object instantiation. Section 5 details how the injector is implicitly made available.

4 Cleaning up instance creation

The design patterns by Gamma *et al.* [GHJV94] are often ways of addressing language limitations. It is not surprising that by introducing a language change as powerful as dependency injection some of the design patterns will become obsolete. A special class of design patterns that we care about in this section are the creational ones, since we have seen in subsection 2.1 that a considerable percentage of static methods are responsible for instance creation.

The abstract factory pattern has been shown to frequently dumbfound users of APIs that make use of it [ESM07]. Gamma defines the intent of the abstract factory pattern as to "provide an interface for creating families of related or dependent objects without specifying their concrete classes" [GHJV94]. Gamma gives the example of a user interface toolkit that supports multiple look and feel standards. The abstract factory pattern then enables code to be written that creates a user interface agnostic to the precise toolkit in use.

Let us suppose the existence of two frameworks A and B, each with implementations of an abstract class Window, named AWindow and BWindow, and the same for buttons. Following the abstract factory pattern, Figure 7 shows how we could create a window with a button that prints "World!" when pressed.

```
createWindow: aFactory
    window :=~(aFactory make: #Window) size: 100 @ 50.
    button :=~(aFactory make: #Button) title: 'Hello'.
    button onClick: [Transcript show: 'World']. window add: button.
```

Figure 7 – Object creation with Abstract Factory

Ellis *et al.* [ESM07] show that using this pattern dumbfounds users. When presented with the challenge of instantiating an instance that is provided by a factory, they do not find the required factory. In Seuss, the code snippet in Figure 8 may generate a window either using framework A or B, depending on the configuration, with no need to find (or even write) a factory:

Seuss allows writing natural code that still bears all the flexibility needed to exchange the underlying framework. It can be used even on code that was not written

```
createWindow
    window := Window size: 100 @ 50.
    button := Button title: 'Hello'.
    button onClick: [Transcript show: 'World'].window add: button.
```

Figure 8 – Replacing object creation with Dependency Injection

with the intention of allowing the change of the user interface framework.

5 Dependency injection as a language feature

Normally, using dependency injection frameworks requires intrusively modifying the way code is written. The developer needs to make the following modifications to the code:

- Add the definition of an instance variable.
- Specify through an annotation which instance variable gets injected (the inject annotation from Figure 4).
- Provide a method through which the dependency injection framework can set the instance variable to the value of the injected object. This is a setter method in Smalltalk and Java (Figure 4) or a dedicated constructor in Java.

To improve usability, in Seuss[4] we completely remove the requirement of modifying the code in any of the previously mentioned ways. As a result, the code in Figure 3 is interpreted just as if the code in Figure 5 and Figure 4 had been written.

The feature that allows us to use dependency injection without the invasive modification of source code is a slight change to the Smalltalk language: for every global being accessed, the access is redirected to an instance variable. This instance variable is annotated for injection, made accessible through setters, and then is set by the framework *when the object is created*.

Since every access to a global is redirected to an instance variable, it is tempting to store the global in the instance variable. However, this will not suffice. That is because the unaltered class is not aware of Seuss and thus does not inject dependencies into objects it newly creates. Instead, we inject an *instantiator object* that resembles the class, but knows the injector and uses it for initializing new instances.

The class of the instantiator is an anonymous copy of the original class, save for the following modifications. First, for every global that a class method calls, a class variable is added, and declared as an injectable dependency. Thus, the instantiator has more class variables than the class it is a copy of. Second, the class methods are recompiled so that all accesses to globals are redirected to their respective class variables. Third, if the original class was not Object, but another class, then the superclass of the instantiator is chosen to be the instantiator of that other class. If no such instantiator exists, one is created. Fourth, by adding a trait [SDNB03] to the instantiator, the instantiator overwrites the basicNew method[5] to inject all dependencies into newly created objects, using the injector.

[4] Seuss can be downloaded at `http://www.squeaksource.com/Seuss.html`

[5] basicNew is a primitive that allocates memory for the new object. It is normally not overridden.

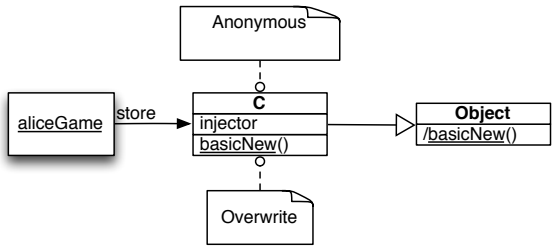


Figure 9 – Class C is injected into object aliceGame. Instances of C mimic Store, but use the injector when creating instances.

For example, in Figure 4, the object that is injected into instance variable store is the anonymous class C. As illustrated in Figure 9, C overwrites method basicNew which is inherited from Object class.

However, not the entire system can be interpreted using the new semantics. That is because Seuss cannot use the features it provides itself. This is the same meta-regression problem that reflective systems encounter, known as the "reflective tower." [TNCC03, p. 13]

In the current implementation, the injector works reflectively. When asked to inject into an object, it analyzes the object's class for declared dependencies, looks them up in the current configuration, and reflectively invokes the appropriate setter to inject each dependency. This is a straightforward and simple implementation, but clearly leaves room for optimization. We will discuss some options in section §8.2.

In order to change the semantics of a standard Pharo as described above, we use Helvetia [RGN10], a language workbench for Smalltalk. Helvetia lets us intercept the compilation of every individual method. Helvetia requires us to specify our language change as a *Rule*, which is really a transformation from one method AST to another. When changing methods, we also modify the containing class when needed. During the transformation, we also create and update a default configuration, which lets the code run as before, if used. It can also be overridden by the user in unit tests. Algorithm 1 details the transformation.

Introducing dependency injection as a language feature brings two advantages:

1. *Backwards compatibility.* Dependency injection can be used for code that was not written with dependency injection in mind. We were able to use the unit test from Figure 6 without having to modify the SandstoneDB project, which does not use dependency injection.

2. *Less Effort.* Other frameworks require that all dependencies be explicitly declared through some boilerplate code for each dependency. In our case, by automatically injecting needed dependencies where possible, the amount of code to write was reduced.

6 Performance

We show that the performance of Seuss is worse than that of standard Smalltalk for identical code, but not prohibitively so. Since Seuss stores all dependencies of an object

Algorithm 1 Transforming ordinary code into dependency injected code.

1. Replace every occurrence of a global with an access to an instance variable. Add that instance variable if necessary.

2. Generate a setter method for that variable and annotate it so that the dependency injection framework can inject into that variable.

3. If the injected global is a class, act as follows. Generate an anonymous metaclass C as described above, and make its instance known to the default configuration. As described above, the instance should behave just like the original class, but should additionally inject all dependencies into newly created instances of class C.

4. Make the default configuration aware of the referred to global.

in instance variables, Seuss objects require more space in memory. Slowdown is due not only to increased memory consumption, but also to the need for instance variables to be set by the injector at instantiation time. Currently, Seuss's implementation to carry out instantiation is straightforward, but reflective at instantiation time, which is slow. We perform benchmarks to assess both memory consumption and time performance, and show that even the current implementation is workable for practical purposes, while there remains much room for optimization.

In our current, naive implementation, Seuss impacts the performance of applications primarily during object instantiation when there is a penalty for injecting all dependencies. The other performance hit is due to pointers to globals being replaced by pointers to instance variables. The resulting increased memory consumption implies a performance hit in itself, since caches will hit less frequently. Further, it makes some optimizations harder, though not impossible. Since the underlying computation is the same, in principle all the above performance penalties can be undone. In Section §8.2 we will see which optimizations a compiler will have to perform to undo the performance hit we cause.

6.1 Evaluation setup

We evaluate the performance implications of having all objects created using dependency injection on the example of the Seaside[6] web framework.

Using Seuss, we run two versions of the Seaside framework. Both are identical in code, but the semantics of one are transformed using Seuss, while the other is executed according to standard Smalltalk semantics. We also transformed all dependencies of Seaside that are not contained in a standard image of Pharo Smalltalk, namely the http server, 'KomHttpServerLight', and 'Grease', a compatibility layer between different dialects of Smalltalk. Since no further modifications are carried out, both bases of code behave the same way, save for their difference in performance and memory consumption.

[6] `http://seaside.st`

As a sample server application, we chose the 'Sushi store', an online shop that ships with Seaside and consists of 16 classes. We script our browser[7] to follow a simple shopping scenario: add three items to the cart, remove one, add another one, then check out and pay for the purchase. Three instances of Firefox are employed in parallel, each running the script three times. The Firefox instances query a server running on the same machine, a 2.3 GHz 2011 Mac Book Pro.[8]

6.2 Memory Consumption evaluation

To observe the effects of Seuss on a large, realistic code base, we analyzed the Seaside framework, together with Pier. Seaside is a commercially and widely used web framework. Pier[9] is a content management system implemented on top of it. When all dependencies of Seaside (excluding the standard Smalltalk libraries, those dependencies are Pier, KomHttpServerLight, Grease, and Magritte) are included into the count, the framework counts 1300 classes. We analyzed these classes, before and after transforming them using Seuss,[10] to establish the extra memory footprint and added execution time.

As can be seen in Table 1, the number of additional instance variables in each class is considerable. The number of inherited variables grows faster than the number of instance variables, simply because it is the sum of all instance variables added in superclasses. Thus, the more superclasses a class has, the more added instance variables it must bear. Altogether, a mean of 13 instance variables is added per class, counting inherited instance variables.

Table 1 – Number of instance variables before and after transforming it with Seuss

		Mean	Std. dev.	Median
Local instance variables	Without Seuss	0.902365	1.76555	0
	With Seuss	2.72006	4.53942	1
Inherited instance variables	Without Seuss	3.30587	2.79563	3
	With Seuss	16.5278	12.0294	13

As a result of the increase in the number of instance variables, the size of the objects of most classes will thus increase by an order of magnitude. However, perhaps unsurprisingly, most space in the system is occupied by arrays and other simple datatype objects, which are unaffected by Seuss. By not transforming simple datatype objects, we can transform almost all classes in the system, and still affect only a minority of all objects.

The last claim can easily be verified in any image of Pharo, which allows one to count all instances of all classes. Figure 10 shows that the number of instances per class falls faster than exponentially. In our test system, the objects of the first 4 classes, Array, Association, ByteString and CompiledMethod, together outnumber all other objects in the system by 1,154,310 to 991,462 objects.

[7] To script the browser, we used the Selenium WebDriver in Ruby, `http://seleniumhq.org/`.
[8] The script can be found here: `http://github.com/nes1983/selenium-bench`.
[9] `http://www.piercms.com/`
[10] The transformation is enabled by changing the default compiler in Pharo. The entire transformation runs in under an hour.

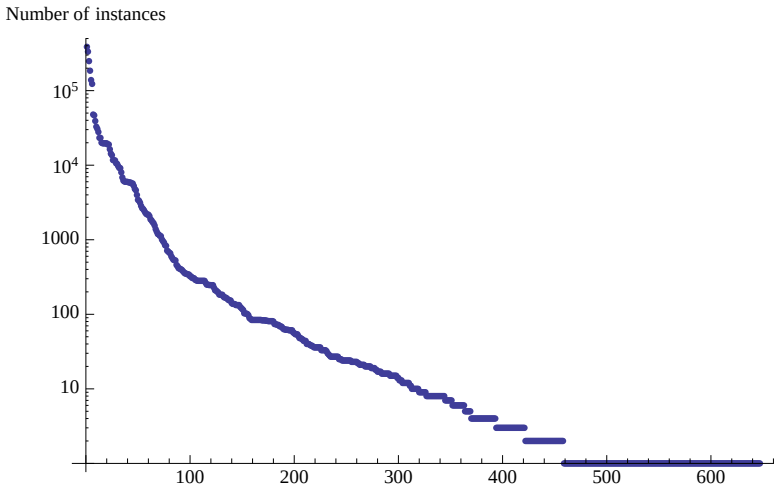


Figure 10 – Measured probability density function of number of instances per class. The y-axis is the number of instances for a class, sorted in descending order.

In order to see the impact of the increased number of instance variables on the total memory consumption, we computed the memory per session after each run of the benchmark. Without Seuss, all sessions were exactly 290,436 bytes in size. After the transformation by Seuss, this size increased to 439,712 bytes, which is an increase of 33.9 %.

Since all sessions had exactly the same size after each run of the benchmark, we could pick one at random to analyze. The size of an object, including all objects visible from it, can be computed in Smalltalk by traversing the object graph and aborting whenever the global scope is reached. This is implemented in the Seaside project.[11] The names of instance variables that Seuss adds, all end in a magic suffix, making them identifiable. The added memory consumption is the number of added instance variables reached during traversal, multiplied by pointer size in bytes. We could compute the additional memory consumption exactly by adapting the traversal of the object graph to count all added instance variables that were reached.

The increase of 33.9 % in memory is incidental to our implementation and compiler, not inherent to the problem. As we will outline in section §8.2, a better implementation with an optimizing compiler could probably execute code according to Seuss's semantics at no additional performance or memory penalty.

6.3 Time Performance evaluation

We ran our benchmark 10 times both in standard Seaside, and in our transformed version of Seaside. As seen in Table 2, the transformed version runs at 51 % of the original performance. To assess if a difference in performance is statistically significant we use the Wilcoxon signed-rank test which we prefer over Student's t-test, because deviations from mean execution times are necessarily skewed, and thus, the assumption of a normal distribution is problematic. The Wilcoxon test rejects the null-hypothesis that there is no underlying difference in means with $p = 10^{-5}$ and $W = 100.0$.

[11] The relevant class is WAMemory

Table 2 – The time performance of the Sushi store with and without Seuss.

	Mean run time	Standard deviation
With Seuss	27.097 s	1.653 s
Without Seuss	13.703 s	1.093 s

Although the slowdown is significant, the benchmarked prototype is unoptimized. Possible optimizations are discussed in Section §8.2.

7 Developing in Seuss

Seuss allows for two *modi operandi*. They differ in what gets injected by default if a class is declared to be a dependency. In *Seuss language mode*, the changed semantics for static method calls are taken advantage of, by injecting an *auto factory*. In *compatibility mode*, the class[12] is injected.

Both modes can be mixed, *i.e.* a configuration can explicitly, for every dependency, state whether an auto factory, or a class should be injected. Further, objects assembled in Seuss language mode can interact with objects assembled in compatibility mode and vice versa. As a third option, some objects can (and must, because of the infinite meta regression problem) be used unmodified. The objects of Seuss itself, and all simple data types, must be used unmodified. We will describe the two modi in turn.

7.1 The Seuss language mode

In the Seuss language mode, the syntax and the object model stay the same, but the semantics of static method calls changes. In contrast to the standard Smalltalk, the following two rules hold.

- Classes do not have class sides. Instead, they have factories. Code that appears to access a class literal, instead accesses a factory that must be injected.
- A class can have more than one factory.

Consider the following example.

```
UBSClientManager#saveClientHassan
    client := Client name: 'Hassan' ; get.
    client save.

ActiveRecord subclass: Client.
Client#save
    store save: self name.
```

Since in the code snippet, Client starts with an uppercase letter, the class UBSClientManager will declare a dependency of name Client. The framework will inject this dependency based on its configuration, but by default will be an *auto factory*. So, in our example, while Client with a capital C looks like a reference to a class, it is really an instance variable containing a factory.

[12] Or rather: something very similar. See Section §5.

To understand auto factories, note that most factories, like most constructors, are simple: they accept values and assign them to instance variables. Since the semantics of a standard factory are so simple, we can make them part of the framework in Seuss. For every class, the system provides an auto factory. Auto factories gather parameters until they are asked to produce (or get) a new object. An auto factory produces new objects in just three steps: (1) a new instance of their produced class (in our example, Client) is allocated, (2) all dependencies are injected into the newly allocated instance and (3) all parameters that the factory has saved up are assigned to instance variables of the newly allocated instance. This design is different from other dependency injection frameworks, where an object is either injected into, or provided by a factory, but never both.

Note that the produced object is given no chance to initialize itself. This is by design: the initialization of an object is the responsibility of its factory. This design makes it easier to realize immutable objects. From the point of view of an object, as soon as any method is called, all instance variables will have been assigned already by the factory. If the object refrains from changing its instance variables later on, all instance variables will have the exact same contents for all invocations of all methods.[13] However, the user is free to write custom-written factories that give objects a chance to initialize themselves.

If the developer wishes to have a name: method that does not blindly assign, but retain the right to reject a parameter if it does not appear in a blacklist, they can do that by subclassing Autofactory as follows:

```
AutoFactory subclass: SecureClientFactory.
SecureClientFactory#name: aString
   blacklist includes: aString
      ifTrue: [(Error message: 'Blacklisted') raise].
    super name: aString.
```

7.2 Compatibility mode

Using Seuss, we can run standard Smalltalk code in such a way that all accesses to class literals and other globals are transformed into accesses to instance variables, gaining configurability. The implementation of this *compatibility mode* is described in Section §5.

To run legacy code, all the user has to do is create a *Compatibility module*, which accepts as an input a list of all classes that should be transformed. The Seaside web service is started in compatibility module as follows:

```
compatibility := CompatibilityConfiguration forClassesMatching: 'Seaside-*'.
injector := Injector configure: compatibility.
adaptor := injector getProviderFor: #WAComancheAdaptor.
adaptor startOn: 8080.
```

Running the second line, the injector will create anonymous replacement classes for all matching classes and store them in a compatibility module. The newly created classes are first injected and then initialized by calling their initialize method.

[13] The case for immutability has often been made, but never as pointed as by Hickey [Hic11] "State complects value and time," which makes it the opposite of simple.

Running code using the compatibility module allows us to replace any configured dependency. For example, the following snippet will replace all accesses to the Transcript by access to a mock object.

```
compatibility := CompatibilityConfiguration forClassesMatching: 'Seaside−*'.
configuration := Configuration configure:
    [:module | module bind: #Transcript toInstance: MockTranscript new].
injector := Injector configure:∼(configuration override: compatibility) .
adaptor := injector getProviderFor: #WAComancheAdaptor.
adaptor startOn: 8080.
```

8 Discussion

In this section, we explore the challenges for implementing Seuss in statically-typed languages like Java. We describe optimizations that can improve performance or entirely eliminate the performance overhead introduced by Seuss. We touch on human factors and security implications.

8.1 Challenges for statically typed languages.

In a language where classes are reified as first-class objects, such as Smalltalk, classes can simply be injected as objects. In other languages, such as Java, a proxy must be used.

Seuss works by replacing access to globals by access to instance variables. In a statically typed language, the question arises what type injected instance variables ought to be. To see if our small language change would be feasible in a typed language, we ported part of Seuss to Java. In the following transformation by JSeuss, our Java version of Seuss,[14] the access to the global Store is replaced by an instance variable store (note the lower case initial letter) of type ICStore.

```
class Before {
    void save() {
        Store.storeObject(this);
    }
}
```

is transformed into

```
class After {
    @Inject
    ICStore store;
    void save() {
        store.storeObject(this);
    }
}
```

The interface ICStore is a generated interface. Our Java transformation generates two interfaces for every class, one for all static methods, and one for all instance methods. The interfaces carry the same name as the class, except for the prefixed upper-case letters IC, or I, respectively. During class load time, all occurrences of type Store are

[14] The version repository of the project can be found at `http://github.com/nes1983/JSeuss`

then replaced by type ICStore, and so with all classes. All new calls on Store return instances of type IStore. On the other hand, existing interfaces are not touched.

The object of type ICStore serves as a proxy for the class ICStore. This is necessary since classes are not first class in Java, and thus cannot be injected directly. To avoid expensive recompilation, we use Javassist to modify all code at the bytecode level, during class load time.

JSeuss enables unit testing the save method above, and offers the same configurability as the Smalltalk version. However, since many developers wish to develop exclusively in an IDE, combining JSeuss with IDEs remains a challenge. We believe this to be true of most bytecode transformation approaches. In our case, the difficulty stems from the interfaces having to be visible to Eclipse's compiler. We currently achieve this by running a background process that automatically adds all needed interfaces to a directory which must be included in Eclipse's path. While full transparency to the developer would be preferable, this proved to be workable.

8.2 Optimizations

The current code base is naive, and could benefit greatly from just a few optimizations. Since dependencies do not change between instantiations, the instantiator could maintain a pre-computed list of dependencies. Furthermore, instead of reflectively invoking the appropriate setter, different instantiators could be pre-compiled, which do not have to use reflection.

Besides our own codebase showing potential for optimizations, we think that the performance penalty that we measured is entirely incidental to the implementation. That is, we believe that a well-optimized implementation together with a highly optimizing compiler should, together, yield the same performance as Smalltalk code interpreted according to the old semantics.

While dependency injection conceptually requires a large number of instance variables, these instance variables can be optimized away by an optimizing compiler in the following way. The key is that the instance variables that receive dependencies are final, *i.e.* are written into only once, during object creation. Modern compilers can already detect final instance variables. If a compiler now sees that the object written into an instance variable is always the same, the compiler can store that object reference in a hidden table outside of the object, rather than inside the object. Then, methods accessing the instance variable can be rewritten to access the table, instead of an instance variable. This optimization can work if there is a unique configuration in the entire system.

If there are more configurations, then each configuration prescribes its own set of values to be injected. Now, the compiler can maintain hidden subclasses, one for each configuration. If within one configuration a new object is created, the compiler can make the object be a member of the appropriate hidden subclass for that configuration. The subclass differs from its superclass in that all accesses to injected instance variables in the source code are compiled to be accesses to the compiler's table outside of the object. Note that in this scheme, objects, through their class pointers, maintain all information that was previously contained in the injected instance variables. This allows objects to be de-optimized for inspection.

Thus, it may be possible to interpret code according to the semantics of Seuss without adding any instance variables at all. We see no fundamental obstacle to running Seuss at full speed.

8.3 Human factors

In a study by Ko *et al.*, the information need that was observed by far the most frequently was "What caused this program state?" [KDV07]. Although Seuss brings additional state to objects, that injected state is immutable and can easily be traced back to the configuration.

A possible concern is that dependency injection may obscure the intent of the code, since it is no longer clear what a name refers to. However, that is the nature of all decoupling. And untangling things is the path toward simplicity. The reduced navigability can be mitigated by a configuration-aware IDE. An IDE should be able to gather all configurations and use them to display which literals are bound to what.

Since Seuss changes the semantics of the programming language, using it in an IDE poses some challenges. For example, the Pharo code browser, the primary tool for editing Smalltalk code, shows the class and instance side of every class.[15] However, using Seuss, one never needs the class side, and instead would prefer to see, for every class, a list of all factories that create objects of this kind. Similarly, the object inspector draws no distinction between instance variables containing injected state and those containing mutable state, whereas the latter would be more interesting to examine.

8.4 Using Seuss to sandbox code

If Object's reflective methods are removed, then all objects can only find other classes through their dependencies or method parameters. Thus, any piece of code from within a configuration that does not include access to the File class prevents that code from reading or writing files. This concept of security by unreachability was described by Bracha [BvdAB+10].

9 Related work

Dependency injection [Fow02, Pra09] is a design pattern that decouples highly dependent objects. Using it involves avoiding built-in methods for object construction, handing it off to framework code instead. It enables testing of components that would ordinarily be hard to test due to side-effects that would be intolerable in unit tests. There are other frameworks that support dependency injection like Google Guice [Van08] and Spring, after which Seuss's dependency injection capabilities are modeled. In contrast to Google Guice and Spring, Seuss turns dependency injection into a language feature that works even on code that was not written with dependency injection in mind. By superficially allowing the use of standard language constructs for object creation while using dependency injection under the hood, Seuss programs look in large parts like conventional source code.

Achermann and Nierstrasz [AN00] note that inflexible namespaces can lead to name clashes and inflexibilities. They propose to make namespaces an explicit feature of the language and present a language called Piccola based on first-class namespaces. First-class namespaces enable a fine degree of control over the binding of names to services, and in particular make it easy to run code within a sandbox. While Seuss sets the namespace of an object at that object's instantiation time, Piccola allows it to be manipulated in the scope of an execution (dynamically) as well as statically. Similarly,

[15] These are loosely analogous to the distinction between static and instance methods in Java.

some mocking frameworks, such as PowerMock,[16] allow re-writing of all accesses to global namespace to access a mock object. Piccola and PowerMock do not attempt to clean up static method responsibilities, but rather add flexibility to their lookup.

Bracha presents the Newspeak programming language [BvdAB+10], which sets the namespace of an object at that object's instantiation time, just like Seuss. However, while Seuss provides a framework that automatically injects individual dependencies into the dependent object during instantiation time, Newspeak leaves this to the developer. Bracha shows that by restricting a module to accessing the set of objects that were passed in during instantiation time, untrusted software can be sandboxed reliably by not passing in the dependencies that it would need to be harmful, such as file system access modules. The same argument holds for Seuss so long as reflection is disabled. While the rewiring of dependencies is a strong suit of dependency injection, and while Newspeak makes it technically possible, the system's design makes it costly in lines of code to run a unit test in a new configuration. By manually searching for a module instantiation that happens in a unit test, we could not find a single unit test in Newspeak that makes use of Newspeak's capabilities to change namespaces.

We have motivated the need for a dependency injection framework by performing an empirical study on the usage of static methods in a large Smalltalk ecosystem. We are not alone in performing this kind of empirical research—recently there has been a surge in the interest of researchers for studying various aspects of software usage and evolution: the occurrence of ripple effects in software ecosystems [RL11], the usage of reflexion in dynamic programming language [CRTR11], or even the adoption of the dependency injection pattern in a corpus of Java systems [YTM08]. While all these approaches present empirical evidence related to certain aspects of software development and evolution, we have made a step further and, informed by the empirical study we performed, we have provided a technical solution that we have also validated.

10 Conclusion

Static methods pose obstacles to the development of tests by hardwiring instance creation. A study of 120 static methods in open-source Smalltalk code shows that out of the 120 static methods, only 6 could not equally well be implemented as instance methods, but were not, thus burdening their caller with the implicit dependency on these static methods.

The dependency injection design pattern offers a partial solution to decoupling responsibilities from static methods, but still entails tedious rewriting of application code and the use of boilerplate code. We have shown how introducing dependency injection as a language feature can drastically simplify the task of migrating class responsibilities to instance methods, while maintaining code readability and enabling the development of tests through explicit configurability. Moreover, we have shown how a language with dependency injection as a feature becomes more powerful and renders certain design patterns obsolete.

We have demonstrated the feasibility of the approach by presenting Seuss, an implementation of dependency injection as a language feature in Smalltalk. We have discussed aspects regarding the usage of Seuss from the perspective of developer usability. We have presented benchmarks that show that Seuss introduces a 34 % additional memory cost, and runs at 53 % speed, without any optimizations. We

[16] `http://code.google.com/p/powermock/`

have furthermore demonstrated the feasibility of our approach in the context of statically-typed languages by presenting JSeuss, a port of Seuss to Java.

References

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *Lecture Notes in Computer Science*, chapter 8, pages 77–89. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2000. URL: http://dx.doi.org/10.1007/10722581_8, doi:10.1007/10722581_8.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. *SIGPLAN Not.*, 39(10):331–344, October 2004. URL: http://dx.doi.org/10.1145/1035292.1029004, doi:10.1145/1035292.1029004.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://portal.acm.org/citation.cfm?id=1884007.

[BW89] R. Wirfs Brock and B. Wilkerson. Object-oriented design: a responsibility-driven approach. *SIGPLAN Not.*, 24:71–75, September 1989. URL: http://dx.doi.org/10.1145/74878.74885, doi:10.1145/74878.74885.

[CRTR11] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: The case of smalltalk. In *Proceedings of the 8th working conference on Mining software repositories (MSR 2011)*, pages 23–32, New York, NY, USA, 2011. IEEE Computer Society. URL: http://scg.unibe.ch/archive/papers/Call11aDynamicFeaturesMSR2011.pdf, doi:10.1145/1985441.1985448.

[ESM07] Brian Ellis, Jeffrey Stylos, and Brad Myers. The Factory Pattern in API Design: A Usability Evaluation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 302–312, Washington, DC, USA, May 2007. IEEE. URL: http://dx.doi.org/10.1109/ICSE.2007.85, doi:10.1109/ICSE.2007.85.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321127420.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994. URL: http://www.aw-bc.com/catalog/academic/product/0,1144,0201633612,00.html.

[Hic11] Rich Hickey. Simple made easy, Oct 2011. URL: http://www.infoq.com/presentations/Simple-Made-Easy.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, May 2007. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ICSE.2007.45, doi:10.1109/ICSE.2007.45.

[KMA04] Andrew J. Ko, Brad A. Myers, and Htet H. Aung. Six Learning Barriers in End-User Programming Systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dx.doi.org/10.1109/VLHCC.2004.47, doi:10.1109/VLHCC.2004.47.

[Leo08] Ramon Leon. SandstoneDb, simple ActiveRecord style persistence in Squeak, http://www.squeaksource.com/SandstoneDb.html, 2008. URL: http://onsmalltalk.com/sandstonedb-simple-activerecord-style-persistence-in-squeak.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Extreme programming examined*, chapter 17, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Pra09] Dhanji Prasanna. *Dependency Injection*. Manning Publications, pap/pas edition, August 2009. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/193398855X.

[RDK07] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, September 2007. URL: http://scg.unibe.ch/archive/papers/Reng07aMagritte.pdf, doi:10.1007/978-3-540-75209-7_8.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In Theo D'Hondt, editor, *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag. URL: http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf, doi:10.1007/978-3-642-14107-2_19.

[RL11] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems (nier). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 904–907, May 2011. URL: http://scg.unibe.ch/archive/papers/Robb11aRipples.pdf, doi:10.1145/1985793.1985940.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour ECOOP 2003 – Object-Oriented programming. volume 2743 of *Lecture Notes in*

Computer Science, chapter 12, pages 327–339. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003. URL: http://dx.doi.org/10.1007/978-3-540-45070-2_12, doi:10.1007/978-3-540-45070-2_12.

[SLN11] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Better class responsibilities through Language-Based dependency injection objects, models, components, patterns. volume 6705 of *Lecture Notes in Computer Science*, chapter 20, pages 276–289. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-21952-8_20, doi:10.1007/978-3-642-21952-8_20.

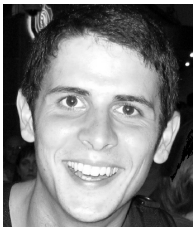
[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, volume 38 of *OOPSLA '03*, pages 27–46, New York, NY, USA, November 2003. ACM. URL: http://dx.doi.org/10.1145/949305.949309, doi:10.1145/949305.949309.

[Van08] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, April 2008. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1590599977.

[YTM08] Hong Yul Yang, E. Tempero, and H. Melton. An empirical study into use of dependency injection in java. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 239 –247, march 2008. doi:10.1109/ASWEC.2008.4483212.

About the authors

Niko Schwarz is a PhD student at the Institute of Computer Science (IAM) of the University of Bern, where he researches into code cloning, dependency injection, and collaboration between developers that are not co-located. Visit him at http://scg.unibe.ch/staff/Schwarz



Mircea Lungu is a PhD researcher at the Institute of Computer Science (IAM) of the University of Bern. His research focuses on the analysis and visualization of software ecosystems in the context of reverse engineering. Visit him at http://scg.unibe.ch/staff/mircea

Oscar Nierstrasz is a Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern, where he founded the Software Composition Group in 1994. Prof. Nierstrasz is co-author of over 200 publications and co-author of the books Object-Oriented Reengineering Patterns and Pharo by Example. Visit him at `http://scg.unibe.ch/staff/oscar`

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper. We thank Simon Vogt for his help in implementing JSeuss. We thank Toon Verwaest and Erwann Wernli for their input.